一种基于偏移寻址的存储高效 IP 地址查找算法

黄昆 谢高岗 李彦彪 刘向阳

摘要 网络带宽的迅猛增长、虚拟路由器和软件路由器等新兴技术的涌现，迫切需要存储高效 IP 地址查找算法。已有的实际 IP 地址查找算法是基于多叉特里树的空间高效编码，例如树位图特里树(Tree Bitmap Trie)。但在这些编码方法中，每个节点维护多个指针以及多个关联的位图，导致特里树的存储空间开销大，难以将信息存储在高速片上存储器中，从而限制了 IP 查找性能。本文提出了一种新颖的偏移编码特里树(Offset Encoded Trie, OET)，实现存储高效 IP 地址查找。偏移编码特里树的每个节点仅维护 1 个下一跳步位图和 1 个偏移值，而不需要孩子指针和下一跳步指针。每个节点利用下一跳步位图和偏移值计算出下一搜索节点的存储地址。在 IP 地址查找过程中，片上偏移编码特里树查找出最长匹配前缀，而片外前缀哈希表查找出与该前缀想关联的下一跳步信息。本文采用实际 IP 前缀规则集进行了实验评估，实验结果表明：与已有多叉特里树编码方法相比，偏移编码特里树显著减少了存储空间开销。

关键词：路由器 IP 地址查找 最长前缀匹配 特里树

1　引言

IP 地址查找是互联网路由器的核心，通常采用最长前缀匹配(Longest Prefix Matching)。即在 IP 前缀规则集中，查找与数据包的目的 IP 地址相匹配的最长前缀规则[1]。在核心路由器中，IP 前缀规则集包含百万多条规则，且每条 IP 前缀规则是由 IP 地址前缀及其关联的下一跳步(Next Hop)信息构成。其中下一跳步信息包括转发端口、转发路由器 MAC[1]地址等。IP 路由器可运行在静态或动态模式下。在静态模式中，路由器采用离线方式周期性更新 IP 前缀规则集，适合快速 IP 查找；在动态模式中，路由器采用在线方式实时更新 IP 前缀规则集，可能中断 IP 查找。凯撒(M. Caesar)等人[2]指出，对 IP 前缀规则集的动态更新策略导致路由性能降低。因此，本文重点研究静态 IP 地址查找算法，支持离线更新操作，实现快速数据包转发。

随着网络带宽和业务流量的迅猛增长，IP 地址查找算法面临可扩展性挑战，即如何满足高速数据包处理的吞吐量和空间需求。IP 地址查找是计算密集型操作，运行在路由器的关键数据路径，已成为路由器的性能瓶颈。近年来，日新月异的网络技术及其应用迫切需要存储高效 IP 地址查找算法。这是因为：

– 互联网的骨干链路带宽从 40Gbps 增至 100Gbps[3-4]，这要求降低 IP 地址查找算法的存储空间需求，从而实现线速 IP 查找。
– IP 前缀规则集随网络规模呈指数级增长，例如当前的核心路由器已包含约 31 万条 IP 前缀规则[5]，为了实现在片上存储器上存储并查找整个 IP 前缀规则集必须压缩 IP 地址查找算法的转发数据结构。
– 日益普及的虚拟路由器技术是一种支持可编程的新兴网络技术，即在单个物理硬件

[1] Media Access Controller，媒体存储控制器

平台上并发运行多个虚拟路由器。每个虚拟路由器独立维护自身的 IP 前缀规则表。可扩展虚拟路由器要求使每个虚拟路由器的存储空间开销最小化，从而支持更多的并发虚拟路由器。

– 业界广泛使用的软件路由器技术是利用多核处理器平台的强大并行计算能力，采用软件方式设计与实现高速路由器。高性能软件路由器要求在片上高速缓存中存储与查找整个 IP 前缀规则集，加速多核处理器的 IP 查找。因此，促使研究者重新关注 IP 地址查找算法的紧凑型转发数据结构，以实现快速和可扩展 IP 查找。

IP 地址查找算法可分为基于 TCAM[2]、基于哈希和基于特里树(Trie)。基于 TCAM 的算法[10-12]可提供确定性和高速 IP 查找，但是存在开销高和能耗高等缺点。基于哈希的算法[3-4, 13-18]利用哈希表加速 IP 查找性能，但是存在存储器带宽需求大等缺点。由于特里树是一种高效数据结构，基于特里树的算法[19-27]广泛应用于高速路由器、防火墙和 NIDS/NIPS[3]等。当前的虚拟路由器和软件路由器主要采用基于特里树的高效 IP 地址查找算法，满足其可扩展性和可编程性等需求。

但是，已有的基于特里树的 IP 地址查找算法[19-23]仍存在存储空间开销大等问题。IP 地址查找算法通常采用多叉特里树表示一组 IP 前缀规则，即一次检查 IP 地址的多个比特，以增加存储空间开销为代价来提高其吞吐量。许多实际 IP 地址查找算法是采用多叉特里树的空间高效编码方法，例如树位图特里树(Tree Bitmap Trie)[22]，从而显著减少其存储空间需求。在这些编码方法中，特里树的每个节点维护孩子指针、下一跳步指针以及关联的位图(Bitmap)。孩子指针指向孩子节点，其大小为 $\log_2 n$ 比特；而下一跳步指针指向下一跳步信息，其大小为 $\log_2 m$ 比特；位图的大小为 $O(2^s)$ 比特，其中 n 表示节点个数，m 表示下一跳步信息个数，s 表示步长的比特个数。由于 n 和 m 随着前缀规则条数的增加而增加，节点的孩子指针和下一跳步指针占用更多存储空间，导致这些特里树编码方法的可扩展性差。例如，树位图特里树的每个节点维护 1 个外部位图及 1 个孩子指针、1 个内部位图及 1 个下一跳步指针；假设 IP 前缀规则集包含 310K 条 IP 前缀规则，二叉树位图特里树包含 219K 个节点；整个树位图特里树占用 15Mb 的存储空间，无法存储在最新的 10Mb 片上 SRAM[4]中，从而限制了 IP 地址查找性能的提高。

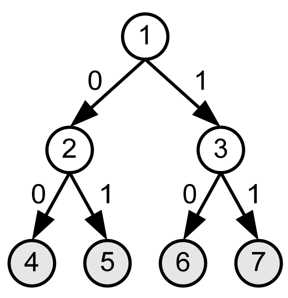


图1. 二叉特里树

为了解决上述问题，本文提出了一种基于偏移寻址(Offset Addressing)的存储高效 IP 地址查找算法，即采用偏移编码特里树(Offset Encoded Trie, OET)表示一组 IP 前缀规则，从而显著减少其存储空间需求。偏移编码特里树(以下简称 OET)的每个节点仅维护 1 个下一跳步位图和 1 个偏移值，而不需要孩子指针和下一跳步指针。下一跳步位图是用于记录孩子节点与最左非叶子(Non-leaf)孩子节点的索引偏移；偏移值是用于记录每个节点与其最左非叶子孩子节点的标识符距离。每个节点利用上述两个偏移值来计算出下一搜索节点的存储地址。在 IP 地址查找过程中，片上 OET 查找出最长匹配前缀，并从片外前缀哈希表中查找出与该前缀关联的下一跳步信息。例如，图 1 给出一颗二叉特里树，包含 7 个节点。树位图特里树的每个节点维护 1 个大小为 2 比特的外部位图及 1 个大小为 3 比特的孩子指针、1 个大小为 1 比特的内部位图及 1 个大小为 2 比特的下一跳步指针，则每个节点的存储空间为 8 比特。

[2] Ternary Content Addressable Memory，三态内容可寻址存储器
[3] Network Intrusion Detection/Prevention System，网络入侵检测与防御系统
[4] Static Random Access Memory，静态随机存储器

而 OET 的每个节点仅维护 1 个大小为 2 比特的下一跳步位图和 1 个大小为 1 比特的偏移值，每个节点的存储空间仅为 3 比特。实验结果表明，与已有的多叉特里树编码方法相比，OET 显著减少存储空间开销。对于实际 IPv4 和 IPv6 前缀规则集，与树位图特里树相比，OET 在存储空间开销上分别减少了 76%和 63%。

本文的组织结构是：第 2 节介绍 IP 地址查找的相关工作；第 3 节详细阐述基于偏移寻址的存储高效 IP 地址查找算法；第 4 节给出实验结果；最后，第 5 节总结全文。

2 相关工作

IP 地址查找算法是互联网路由器性能的关键，已成为网络技术研究热点问题。近年来，研究者提出了许多 IP 地址查找算法，可分为 3 类：基于 TCAM、基于哈希和基于特里树。

TCAM 是一种内容寻址存储器，即根据内容查找出与其匹配的地址索引。当前的高端路由器主要采用基于 TCAM 的 IP 地址查找算法，即在 1 个时钟周期内完成 1 次 IP 查找，可提供确定性和高速 IP 地址查找。但是，基于 TCAM 的算法存在能耗大、代价高和密度低等问题，难以大规模扩展应用。近年来，研究者提出了多种能量高效和存储高效的基于 TCAM 的算法[10-12]，降低 TCAM 能耗和存储开销，从而提高其 IP 查找性能。与 TCAM 相比，SRAM 在访问速率、存储密度和能耗等方面更加优越。因此，基于 SRAM 的 IP 地址查找算法可替代基于 TCAM 的算法，实现低能耗的高速 IP 查找。

基于哈希的 IP 地址查找算法[3-4, 13-18]是在片上 SRAM 中采用哈希表来实现线速 IP 地址查找。近年来，研究者采用布隆过滤器(Bloom Filter)及其变型 [3, 14, 16, 18]，以减少哈希表的片外 DRAM[5]访问次数，从而加速基于哈希的 IP 地址查找算法，其吞吐量可达 100Gbps。但是，基于哈希的算法存在存储器访问带宽需求大等问题，即需要代价昂贵的多端口高速存储器来支持 IP 地址查找的并行处理。

基于特里树的 IP 地址查找算法是最流行的数据包处理算法，广泛应用于交换机/路由器、防火墙、NIDS/NIPS 等关键网络设备中。近年来，研究者提出了许多基于特里树的新算法[19-24]来提高存储效率和查找吞吐量。但是，这些基于特里树的算法性能随着树深度的增加而线性降低，导致无法线速处理 IP 数据包。为了提高基于特里树的 IP 查找吞吐量，研究者提出了存储器流水线(Memory Pipeline)方法[25-27]，即在 1 个时钟周期内完成一次 IP 地址查找。由于片上存储器的空间小且价格昂贵，存储器流水线的每个阶段需要空间高效的特里树数据结构，以利于实现多阶段和多流水线之间的存储高效和负载均衡。

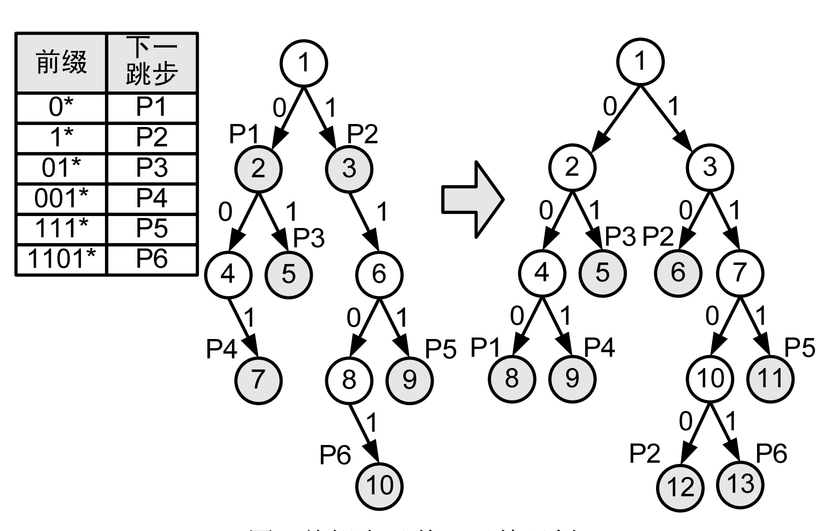
多核处理器技术是在一块芯片上集成多个并行 CPU 核，从而提升整个芯片的计算处理能力。为了提高路由器的可扩展性和可编程，多核处理器技术广泛应用于虚拟路由器和软件路由器，要求存储高效的特里树数据结构，从而以软件方式实现高速 IP 地址查找。近年来，研究者提出了特里树重叠(Trie Overlay)和特里树编织(Trie Braiding)等方法，构建多个虚拟路由器的紧凑型共享特里树数据结构。与这些研究工作相补，本文的偏移编码方法用于压缩单个虚拟路由器的特里树数据结构，从而减少多个虚拟路由器的整个存储空间开销，有助于构建可扩展的虚拟路由器。

3 偏移编码特里树

[5] Dynamic Random Access Memory，动态随机存储器

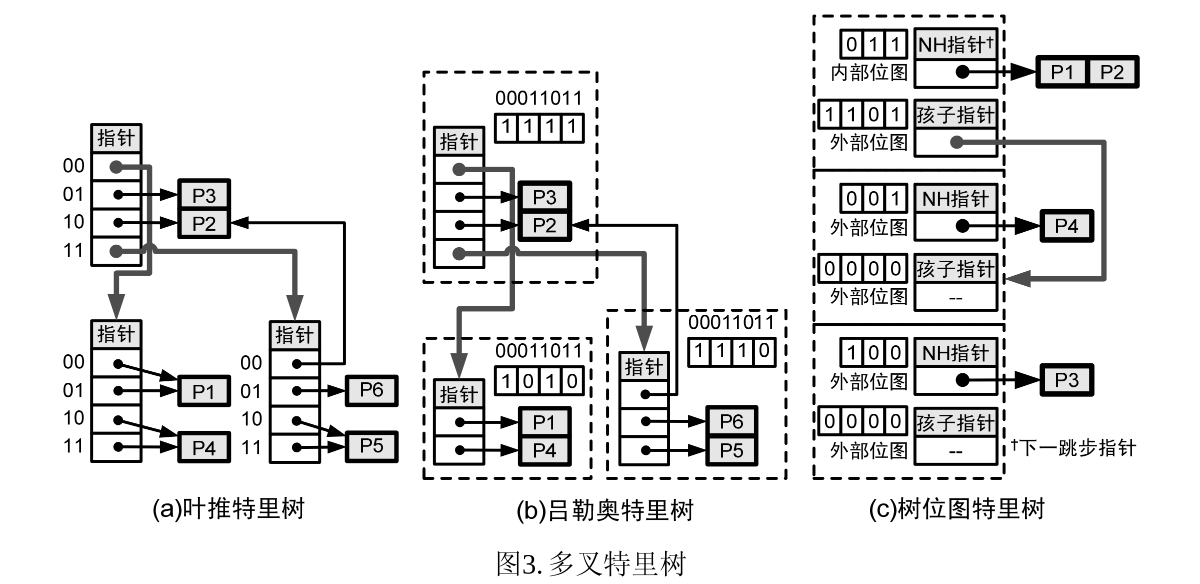
3.1 已有特里树编码方法

二叉特里树是一种基本的树型数据结构，应用于 IP 地址查找。特里树数据结构是用于表示一组 IP 前缀，且采用一个节点表示每个匹配前缀，称为前缀节点(Prefix Node)。在特里树中，每个 IP 前缀值对应于一条从根节点到前缀节点的路径。当查找 IP 地址时，从根节点遍历特里树，直到匹配出最长前缀。在二叉特里树的遍历过程中，IP 地址的连续比特决定遍历路径，即比特为 0 表示遍历左孩子，比特为 1 表示遍历右孩子。图 2 给出了一个前缀表及其二叉特里树(左边)，其中阴影节点表示前缀节点。在二叉特里树中，每个节点包含 1 个左孩子指针、1 个右孩子指针和 1 个下一跳步指针。当查找 IP 地址 1101 时，虽然阴影节点 3 匹配出前缀 1*，但是后续的特里树遍历匹配出阴影节点 10 的最长前缀 1101*，从而输出对应的下一跳步信息 ***P***6 。



前缀	下一跳步
0*	P1
1*	P2
01*	P3
001*	P4
111*	P5
1101*	P6

图2. 前缀表及其二叉特里树



(a)叶推特里树 (b)吕勒奥特里树 (c)树位图特里树

图3. 多叉特里树

叶推特里树(Leaf-Pushed Trie)[20]是一种最流行的特里树变型，即将中间节点的前缀向下推至叶子节点上，确保所有前缀节点为叶子节点。图 2 右边给出了对应的二叉叶推特里树。例如，左边特里树中中间节点 3 的前缀 1*推至右边特里树中叶子节点 6 和叶子节点 12，其对应的扩展前缀分为 10*和 1100*。在步长为 ***s*** 的叶推特里树中，每个节点包含 2^s 个指针项，其中每个指针项包含 1 个孩子指针或 1 个下一跳步指针。图 3(a)给出了一个步长为 2 的叶推特里树，其中每个节点包含 4 个指针项。在该叶推特里树中，孩子指针大小为 2 比特，下一跳步指针大小为 3 比特。例如，根节点的第$(01)_2$和第$(10)_2$项分别包含指向 ***P***3 和 ***P***2 的下一跳步指针，而其他指针项包含孩子指针。因此，由于每个指针项要么包含孩子指针，要么包

含下一跳步指针，与原始的特里树相比，叶推特里树减少约一半的存储空间开销。

吕勒奥特里树(Lulea Trie)[21]和树位图特里树(Tree Bitmap Trie)[22]是空间高效的多叉特里树编码方法。吕勒奥特里树是一种叶推特里树的变型，即每个节点利用 1 个位图来表示连续的重复指针，从而消除大量的冗余指针。图 3(b)给出了一个步长为 2 的吕勒奥特里树，其中每个节点包含 1 个大小为 4 比特的位图，比特为 1 表示唯一指针，比特为 0 表示重复指针。例如，左下节点仅包含 2 个下一跳步指针，分别指向 ***P***1 和 ***P***4，以及 1 个位图 1010。树位图特里树是一种非叶推特里树的变型。在步长为 ***s*** 的树位图特里树中，每个节点包含 1 个大小为 2^s 比特的外部位图(EBMP)及 1 个孩子指针、1 个大小为 2^s-1 比特的内部位图(IBMP)及 1 个下一跳步指针。图 3(c)给出了一个步长为 2 的树位图特里树，其中每个节点包含 1 个大小为 4 比特的外部位图及 1 个大小为 2 比特的孩子指针、1 个大小为 3 比特的内部位图及 1 个大小为 3 比特的下一跳步指针。例如，根节点包含 1 个下一跳步头指针指向 ***P***1 和 ***P***2 的链接列表、1 个孩子头指针指向连续孩子节点的列表，以及设置外部位图和内部位图分别为 1101 和 011。

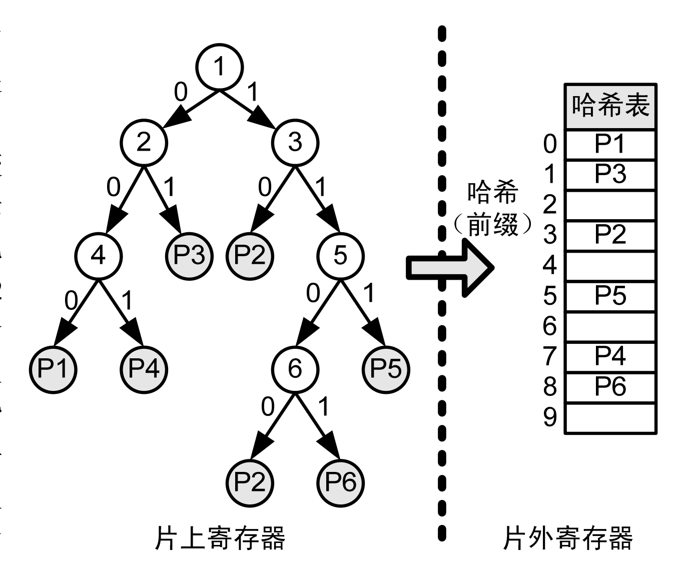


图4. 基于偏移寻址的 IP 地址查找体系结构

与上述特里树编码方法不同，本文采用快速路径和慢速路径相分离的思想。即利用片上高速存储器来减少片外慢速存储器访问次数，从而实现高速 IP 地址查找。已有特里树编码方法通常采用一个片上指针指向一个片外下一跳步信息，导致片上存储器空间需求大。本文采用一个大小仅为 1 比特的标记(Flag)替代下一跳步指针，标识出下一跳步信息是否存在。因此，本文的偏移编码方法可显著减少特里树的存储空间开销，整个存储都可以放在片上存储器中，从而提高 IP 地址查找吞吐量。

本文提出了一种由片上 OET 和片外前缀哈希表构成的基于偏移寻址的 IP 地址查找体系结构。OET 存储在片上存储器 SRAM 中，查找出与 IP 地址相匹配的最长前缀；前缀哈希表存储在片外存储器 DRAM 中，查找与该前缀相关联的下一跳步信息。图 4 给出了基于偏移寻址的 IP 地址查找体系结构示意图。如图 4 所示，假设查找 IP 地址 0010，遍历片上 OET，查找出最长匹配前缀 001*；以 001 为关键值，采用哈希函数 ***Hash***(001) 查找片外前缀哈希表，输出与 001*关联的下一跳步信息 ***P***4 。

3.2 二叉偏移编码特里树

偏移编码特里树(OET)是一种叶推特里树的紧凑型编码。在构建 OET 之前，本文提出了一种特里树节点命名方法，即采用“自顶向下从左向右”(Top-down-left-right)顺序来命名节点标识符。在特里树叶推之后，首先命名根节点的标识符为 1，接着按照从左向右的顺序命名该节点的非叶子节点，并依次递归命名。由于孩子节点包含在其父亲节点的数据结构中，该命名方法不需要命名叶子节点。实质上，特里树节点命名方法是一种广度优先遍历方法，用于减少 OET 节点的偏移值大小。图 4 给出了叶推特里树节点命名示例，其中采用“自顶

向下从左向右”顺序来命名非叶子节点标识符，采用下一跳步信息来命名叶子节点标识符。

在二叉叶推特里树中，每个节点包含 3 个字段：左孩子节点标识符、右孩子节点标识符、左右孩子的下一跳步标志。左右孩子的下一跳步标志是用于表示左右孩子是否是前缀节点。如果左孩子或右孩子是前缀节点，左孩子或右孩子的下一跳步标志为 1；否则，左孩子或右孩子的下一跳步标志为 0。图 5 给出了二叉叶推特里树节点数据结构(左边)。根节点 1 包含左孩子节点标识符 2、右孩子节点标识符 3 以及左右孩子的下一跳步标志 00。由于二叉叶推特里树包含共 6 个非叶子节点(如图 4 所示)，孩子节点标识符大小为 3 比特。因此，该二叉叶推特里树的每个节点占用 8 比特片上存储空间。注意：在本文中所有前缀节点的下一跳步信息都存储在片外存储器中。

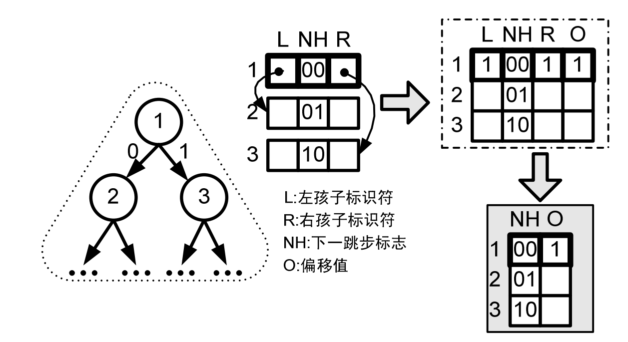


图5. 二叉特里树节点数据结构的转换过程

基于上述二叉叶推特里树，本文采用偏移编码方法构建一个基本二叉 OET。在该 OET 中，每个节点包含 4 个字段：左孩子节点标志、右孩子节点标志、偏移值、左右孩子的下一跳步标志。本文采用左右孩子节点标志来替代左右孩子节点标识符，表示左右孩子是否存在。偏移值是表示节点与其非叶子孩子节点的标识符距离。图 5 点划线框内给出了基本二叉 OET 节点数据结构。根节点 1 设置左右孩子节点标志均为 1、偏移值为 1、左右孩子的下一跳步标志为 00。由于基本二叉 OET 的最大偏移值为 2(如图 4 所示)，偏移值大小为 2 比特。因此，该基本二叉 OET 的每个节点仅占用 6 比特存储空间。

为了进一步减少存储空间开销，本文构建一个高级二叉 OET。在该 OET 中，每个节点包含 2 个字段：左右孩子的下一跳步标志和偏移值。在叶推特里树中，每个非叶子节点均包含左右孩子节点，而每个叶子节点均不包含任何孩子节点。我们观察到：在基本二叉 OET 中，左右孩子的下一跳步标志与左右孩子节点标志是互补的，采用左右孩子的下一跳步标志来替代左右孩子节点标志，用于指出左右孩子是否存在。因此，高级二叉 OET 消除了基本二叉 OET 节点的左右孩子节点标志，而仅保留左右孩子的下一跳步标志。图 5 阴影框给出了高级二叉 OET 节点数据结构。根节点 1 设置左右孩子的下一跳步标志为 00 和偏移值为 1。当查找比特为 0 时，根节点 1 检查出左孩子的下一跳步标志为 0，表示左孩子存在且不是前缀节点，则继续 查找其左孩子节点 2。当查找比特为 1 时，根节点 1 检查出右孩子的下一跳步标志为 1，表示右孩子存在且不是前缀节点，则继续查找其右孩子节点 3。因此，该高级二叉 OET 的每个节点仅占用 4 比特存储空间，而远小于二叉叶推特里树的 8 比特。

地址	左标志	右标志	偏移	下一跳步标志
1	1	1	1	00
2	1	0	2	01
3	0	1	2	10
4	0	0	0	11
5	1	0	1	01
6	0	0	0	11

地址	偏移	下一跳步标志
1	1	00
2	2	01
3	2	10
4	0	11
5	1	01
6	0	11

图6. 二叉偏移编码特里树

图 6 给出了基本和高级二叉 OET 节点数据结构，其中左边是基本二叉 OET，右边是高级二叉 OET。在基本二叉 OET 中，每个节点包含 4 个字段，其大小为 6 比特，则所有节点的总存储空间大小为 6×6=36 比特。在高级二叉 OET 中，每个节点包含 2 个字段，其大小为 4 比特，则所有节点的总存储空间大小为 6×4=24 比特。如图 5 所示，二叉叶推特里树的总

存储空间大小为 6×8=48 比特。因此，与二叉叶推特里树和基本二叉 OET 相比，图 6 的高级二叉 OET 在存储空间开销上分别减少 1/2 和 1/3。

假设 IP 地址为 0101，从根节点 1 查找图 6 中的高级二叉 OET。对于 IP 地址的第 1 位比特 0，根节点 1 查找出左孩子的下一跳步标志为 0，则依据偏移值为 1 和其存储地址为 1，计算出左孩子节点的存储地址为 2，且继续查找左孩子节点 2；对于 IP 地址的第 1 位比特 1，节点 2 查找出右孩子的下一跳步标志为 1，表示右孩子是前缀节点，则查找结束，输出最长匹配前缀 01*。

3.3 多叉偏移编码特里树

互联网路由器通常采用多叉特里树来加速 IP 地址查找吞吐量。但是，在多叉特里树中，节点大小随步长的增加而指数增加，导致特里树的整个存储空间开销快速增大。特别是，当多叉特里树的步长太大时，节点大小的增加会超过节点个数的减少，导致多叉特里树的性能反而降低。研究者已提出了动态规划算法[20]来使多叉叶推特里树的存储空间开销最小化。图 7 给出了叶推特里树的扩展过程，其中左边是二叉叶推特里树，右边是步长为 2 的多叉叶推特里树。如图所示，左边特里树的前缀节点 ***P***1、 ***P***4 和 ***P***5 向下扩展为右边特里树的叶子节点。

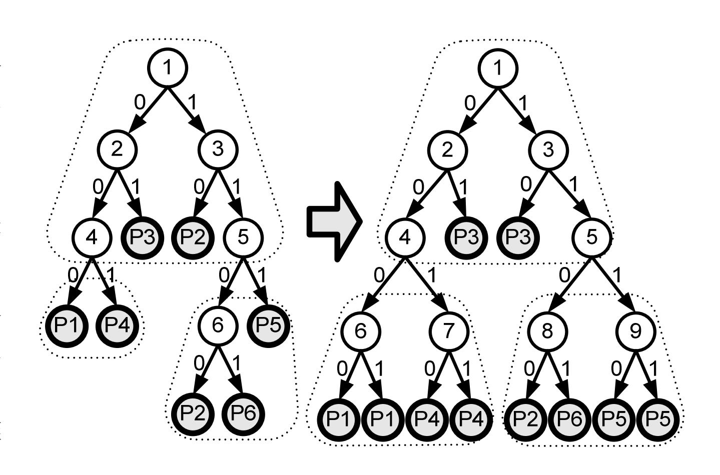


图7. 叶推特里树的扩展过程

与二叉 OET 的构建相似，本文采用偏移编码方法在多叉叶推特里树的基础上构建一个多叉 OET。首先，二叉叶推特里树扩展为步长为 ***s*** 的多叉叶推特里树。在基本多叉 OET 中，每个节点包含 1 个孩子位图(Child Bitmap，CBMP)、1 个偏移值和 1 个下一跳步位图(Next Hop Bitmap，NBMP)。偏移值是表示节点与其最左(Leftmost)非叶子孩子节点的标识符距离。在高级二叉 OET 中，每个节点仅包含 1 个下一跳步位图和偏移值。其根本原因是：孩子位图与下一跳步位图是互补的，采用下一跳步位图来替代孩子位图，表示相应的孩子是否存在。

图 8 给出了基本和高级多叉 OET 节点数据结构，其中左边是基本多叉 OET，右边是高级多叉 OET。在基本多叉 OET 中，每个节点包含下一跳步位图、孩子位图和偏移值；而在高级多叉 OET 中，每个节点仅包含下一跳步位图和偏移值。如图 8 所示，下一跳步位图和孩子位图大小均为 4 比特，而偏移值大小为 1 比特。因此，基本多叉 OET 的总存储空间大小为 3×9=27 比特，而高级多叉 OET 的总存储空间大小为 3×5=15 比特。图 8 表明，与基本多叉 OET 相比，高级多叉 OET 减少约一半的存储空间开销。

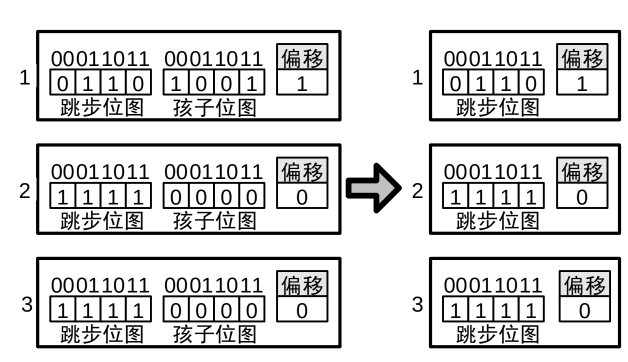


图8. 多叉偏移编码特里树

假设 IP 地址为 1110，查找图 8 中步长为 2 的高级多叉 OET。对于 IP 地址的前 2 位比

特 11，根节点 1 检查出 ***NBMP***[11] 为 0，表明继续查找其孩子节点。在下一跳步位图中，计算第$(11)_2$=3 位之前的 0 个数为 1，且偏移值为 1、节点存储地址为 1，则查找孩子节点的存储地址为 1+1+1=3。对于 IP 地址的后 2 位比特 10，节点 3 检查出 ***NBMP***[10] 为 1，表示匹配成功且查找结束，并输出最长匹配前缀 111*。最后，以 111 为哈希函数的输入参数，查找片外前缀哈希表，输出与前缀 111*关联的下一跳步信息 ***P***5 。

Algorithm: Multi-bit Offset Encoded Trie Lookup

```
Search Offset Encoded Trie (ip_addr)
oet is the multi-bit Offset-Encoded Trie;
s is the stride size;
current_node = oet.getroot();  // get the root node
for (i = 1; i <= len(ip_addr)/s; i++) do
      bit_substr = ip_addr.getbits(i, s);  // get the bit substring
      j = 2^bit_substr-1;  // calculate the index value
      if (current_node.nbmp[j] == 1) do
        match_len = i*s;  // calculate the length of matching prefix
      return ip_addr[match_len];
  else // it continues to search a child node
      index_offset = popcount0s(current_node.nbmp, j);
      child_location = current_node.location
                  + current_node.offset + index_offset;
      current_node = oet.getnode(child_location);
  end do
end do
```

图9. 多叉 OET 的查找算法伪代码

OET 的查找过程与树位图特里树类似，即从根节点开始迭代查找，直至叶子节点。在每一步中，读取 IP 地址的 s 位比特，查找 OET 的节点是否最长匹配，并决策出查找路径。在每个遍历的 OET 节点中，如果 ***NBMP***$[2^s-1]$ 为 1，表示匹配成功，并输出最长匹配前缀；否则，继续查找第 2^s-1 个孩子节点。假设节点的存储地址为 $\boldsymbol{Location}$ ，计算 ***NBMP*** 中第 2^s-1 位之前的 0 个数为 $\boldsymbol{Index}$ ，且其偏移值为 $\boldsymbol{Offset}$ ，则第 2^s-1 个孩子节点的存储地址为 $\boldsymbol{Location}+\boldsymbol{Offset}+\boldsymbol{Index}$ 。多叉 OET 的查找算法伪代码如上图所示。

3.4 变步长偏移编码特里树

本文将上述固定步长 OET 推广到变步长 OET。对于一组前缀规则，斯里尼瓦桑（V. Srinivasan）等人[20]提出了固定步长和变步长叶推特里树的存储空间最小化方法。变步长 OET 是一种变步长叶推特里树的紧凑型编码。因此，变步长 OET 不仅继承了变步长叶推特里树的优点，而且进一步压缩其存储空间开销。变步长 OET 的构建过程与固定步长 OET 的类似。在变步长 OET 中，每个节点仅包含 1 个大小不同的下一跳步位图和 1 个偏移值。图 10 给出了一个最大步长为 2 的变步长 OET 及其节点数据结构。根节点 1 和节点 3 维护 1 个大小为 4 比特的下一跳步位图，而节点 2 维护 1 个大小为 2 比特的下一跳步位图。

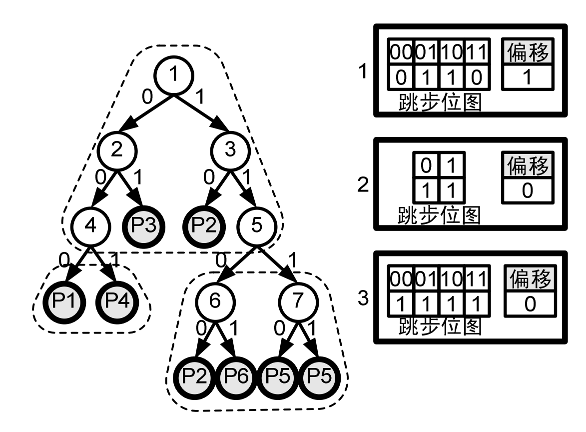


图10. 变步长偏移编码特里树

变步长 OET 的查找过程也与固定步长 OET 的类似。假设 IP 地址为 0010，查找图 10 中的变步长 OET。对于 IP 地址的前 2 位比特 00，根节点检查出 ***NBMP***[00] 为 0，于是依据偏移值为 1 和节点存储地址为 1 计算出下一搜索节点的存储地址为 1+1+0=2。对于 IP 地址的下一比特为 1，节点 2 查找出 ***NBMP***[1] 为 1，表示匹配成功，并输出最长匹配前缀为 001*。最后，以 001 为哈希函数的输入参数，从片外前缀哈希表中查找出与该前缀关联的下一跳步信息 ***P***4 。

3.5 前缀哈希表的构建方法

在本文的 IP 地址查找体系结构中，片外前缀哈希表是用于查找出与最长匹配前缀关联的下一跳步信息。前缀哈希表的性能将会直接影响 IP 查找吞吐量和存储空间开销。近年来，研究者提出了许多高效哈希表，例如快速哈希[30]、孔雀哈希表[31]、布谷鸟哈希表[32]和一次移动哈希表[33]等，以降低存储空间开销并减少哈希冲突概率。这些研究工作可应用于本文的前缀哈希表的设计与实现。

本文采用一种简单高效的多选择哈希方法。该方法广泛应用于基于哈希的 IP 地址查找[3, 13-14]。多选择哈希表采用 $\boldsymbol{k} \geq 2$ 个相互独立的哈希函数，且每个哈希桶存储 ***n*** 个形式为 $\{\boldsymbol{prefix}, \boldsymbol{next\ hop}\}$ 元素。当存储元素时，采用 ***k*** 个哈希函数将前缀映射到 ***k*** 个哈希桶，并选择其中负载最小的哈希桶来存储该元素。当查找元素时，采用相同的 ***k*** 个哈希函数将前缀映射到 ***k*** 个哈希桶，并行查找该 ***k*** 个哈希桶，在每个哈希桶中，顺序查找与最长匹配前缀关联的下一跳步信息。多端口存储器或多并行存储模块等硬件可加速前缀哈希表的并行访问速率。

与已有研究工作[3, 14]类似，本文依据前缀长度将前缀规则集划分为多个组，每组前缀的长度相同，且存储在一个多选择哈希表中。实际上，前缀哈希表是由一组片外多选择哈希表构成。与[3, 14]不同，本文的片上 OET 仅产生一个精确匹配的最长前缀，因此本文选择一个片外多选择哈希表查找下一跳步信息，而不需要查找所有片外多选择哈希表。

3.6 偏移编码特里树的离线更新

网络拓扑变化或瞬时路由失效等将导致 IP 路由表的频繁更新。路由器的控制平面计算出新前缀及其下一跳步信息，并更新线卡(Line Card)的 IP 地址查找数据结构。近年来，凯撒等人[2]指出，动态更新实时响应路由变化将会导致网络资源消耗过多以及更新风暴等问题，甚至中断正常的路由转发。因此，本文采用离线更新操作，即以备份方式更新整个片上 OET，而不会中断数据包转发或产生错误路由等。

IP 前缀规则的增量更新包括下一跳步或前缀的更新。当更新下一跳步信息时，仅需要更新片外前缀哈希表的相应哈希桶，而不需要更新片上 OET。当插入和删除一个前缀时，需构建一个备份的片外叶推特里树，并根据需要来更新片上 OET 和片外前缀哈希表。当删除一个前缀时，仅需要更新片外前缀哈希表中对应的下一跳步信息，而不需要更新片上 OET。前缀插入操作比前缀删除操作更复杂。当插入一个前缀而不创建新的叶推特里树节点时，仅需要更新片外前缀哈希表，而不需要更新片上 OET。当插入一个前缀且创建新的叶推特里树节点时，首先更新备份叶推特里树，接着重新构建整个片上 OET，并依据扩展的前缀来更新片外前缀哈希表。这样，OET 的离线更新操作就可确保无中断、正确的 IP 地址查找。此外，如何确定合适的离线更新周期是我们未来的研究工作。

4　实验评估

本文采用实际 IP 前缀规则集开展模拟实验，以评估基于 OET 的 IP 地址查找算法性能。在评估实验中，本文采用 C/C++设计与实现了四种已有多叉特里树编码方法，即原始特里树、叶推特里树、吕勒奥特里树和树位图特里树等。本文对 OET 与上述四种编码方法的比较集中在存储空间开销方面。

表 1 IP 前缀个数与特里树节点个数

数据集	IPv4		IPv6	
	AS6447	AS65000	AS2	AS1221
前缀数	310344	217952	2259	932
步长	节点数		节点数	
1	500867	348154	8135	3275
2	219406	161282	3655	1461
3	132503	101703	2673	1067
4	82348	66659	1484	584
5	76444	59598	1916	776
6	41520	36813	1752	725

为了评估 IP 地址查找性能，本文选取了四种典型的实际 IP 前缀规则集[5]：AS6447、AS65000、AS2 和 AS1221。其中，AS6447 和 AS65000 是 BGP 路由器的大规模 IPv4 前缀规则集，分别包含约 3.1 万和 2.1 万条 IP 前缀规则；AS2 和 AS1221 是 BGP 路由器的小规模 IPv6 前缀规则集，分别包含约 2 千和 9 百条 IP 前缀规则。表 1 给出了叶推之前的 IP 前缀规则集的具体前缀规则条数及其特里树节点个数。

4.1 IPv4 前缀规则集

实际 IPv4 前缀规则集 AS6447 和 AS65000 分别包含 310344 和 21795 条 IP 前缀规则。图 11 给出了 AS6447 和 AS65000 的 IPv4 前缀长度分布。IPv4 前缀长度分布在 8 至 32 之间。前缀长度为 24 的 IPv4 前缀规则占大多数，例如在 AS6447 中约占 51%，而在 AS65000 中约占 35%。

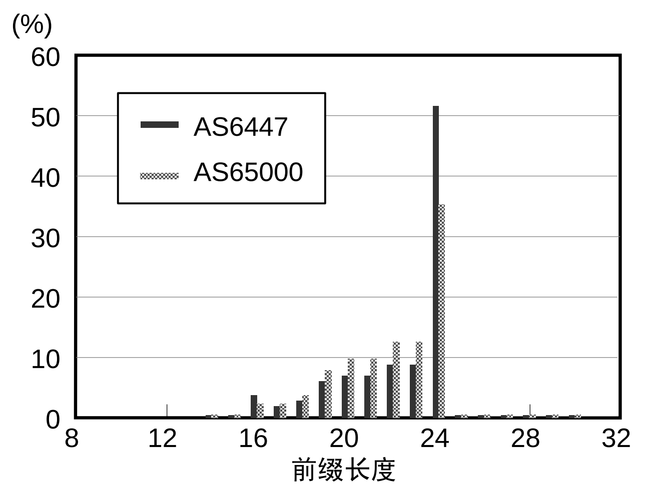


图11. IPv4 前缀长度分布

对于 IPv4 前缀规则集，本文主要给出了在固定步长的情况下不同特里树编码方法的存储空间开销比较。表 1 给出了不同步长的特里树节点个数。当步长大小从 1 增至 6 时，AS6447 的特里树节点个数从 500K 减至 41K，而 AS65000 的特里树节点个数从 348K 增至 36K。在变步长的情况下，本文得出与固定步长的相似实验结果。

图 12 给出了 IPv4 前缀规则集的存储空间开销。如图 12 (a)所示，对于 IPv4 前缀规则集 AS6447，当步长大小从 1 增至 6 时，OET 的存储空间开销仅为 2.51Mb~9.08Mb；与原始特里树、叶推特里树、吕勒奥特里树和树位图特里树相比，OET 在存储空间开销上分别减少了 75%~96.4%、52.5%~93.8%、54.8%~76.6%和 59.3%~77.1%。如图 12 (b)所示，对于 IPv4 前缀规则集 AS65000，当步长大小从 1 增至 6 时，OET 的存储空间开销为 2.03Mb~5.98Mb；与上述其他特里树相比，OET 在存储空间开销上分别减少了 75.7%~96.4%、55%~93.5%、57.1%~73%和 61.1%~74.9%。

总之，对于实际 IPv4 前缀规则集，在不同步长的情况下，与已有特里树编码方法相比，OET 存储空间开销显著减少。

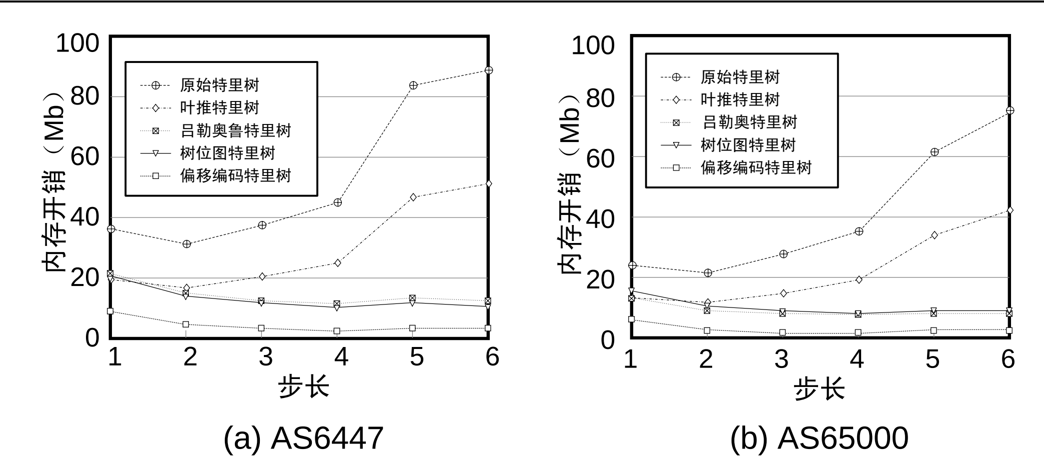


图12. IPv4 前缀规则集的存储空间开销

4.2 4.2 IPv6 前缀规则集

由于当前的 IPv6 网络规模小，实际 IPv6 前缀规则集 AS2 和 AS1221 分别仅包含 2259 和 932 条 IP 前缀规则。图 13 给出了 AS2 和 AS1221 的 IPv6 前缀长度分布。IPv6 前缀长度分布在 16 至 64 之间。前缀长度为 32 的 IPv6 前缀规则占大多数，例如在 AS2 中约占 63%，而在 AS1221 中约占 66%。

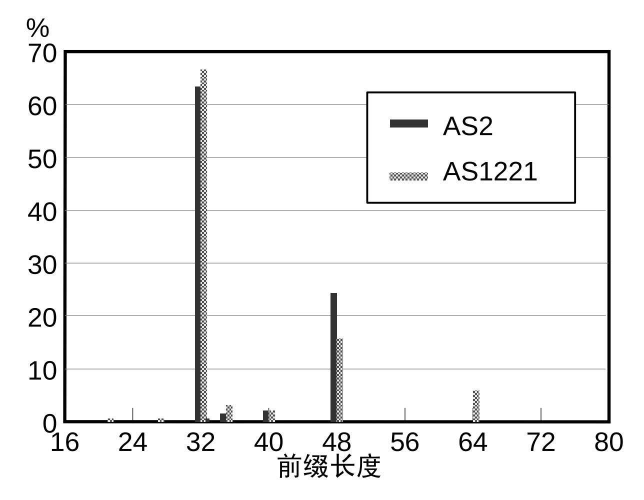


图13. IPv6 前缀长度分布

类似地，对于 IPv6 前缀规则集，本文也主要给出了在固定步长的情况下不同特里树编码方法的存储空间开销比较。表 1 给出了不同步长的特里树节点个数。当步长大小从 1 增至 6 时，AS2 的特里树节点个数从 8K 减至 1K，而 AS1221 的特里树节点个数从 3K 增至 0.7K。在变步长的情况下，本文得出与固定步长的相似性能收益。

图 14 给出了 IPv6 前缀规则集的存储空间开销。如图 14 (a)所示，对于 IPv6 前缀规则集 AS2，当步长大小从 1 增至 6 时，OET 的存储空间开销为 38Kb~125Kb；与原始特里树、叶推特里树、吕勒奥特里树和树位图特里树相比，OET 在存储空间开销上分别减少了 74%~94.9%、53.6%~91%、32.9%~59.9%和 55.4%~62.4%。如图 14 (b)所示，对于 IPv6 前缀规则集 AS1221，当步长大小从 1 增至 6 时，OET 的存储空间开销为 14Kb~51Kb；与原始特里树、叶推特里树、吕勒奥特里树和树位图特里树相比，OET 在存储空间开销上分别减少了 72.7%~94.2%、53.8%~89.5%、29.9%~60.5%和 55%~62.5%。

总之，对于 IPv6 前缀规则集，在不同步长的情况下，与已有特里树编码方法相比，OET 也显著减少存储空间开销。图 14 表明，与原始特里树、叶推特里树、吕勒奥特里树和树位图特里树相比，OET 在存储空间开销上分别平均减少了 73%~95%、53%~90%、31%~60%和 55%~63%。

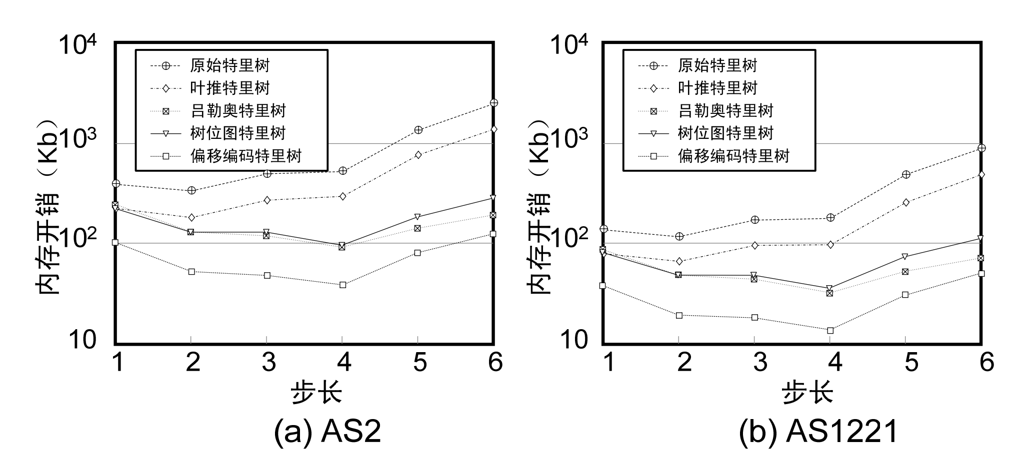


图14. IPv6 前缀规则集的存储空间需求

5 结论

本文提出了一种基于偏移寻址的存储高效 IP 地址查找算法，即采用一种偏移编码方法表示特里树数据结构，称为偏移编码特里树(OET)。OET 的每个节点仅维护 1 个下一跳步位图和 1 个偏移值，而不需要孩子指针和下一跳步指针。当查找 OET 时，每个节点利用下一跳步位图和偏移值来判决查找是否成功，并计算出下一搜索节点的存储地址。实质上，OET 是一种多叉特里树的紧凑等价变型。在 IP 地址查找过程中，片上 OET 查找出最长匹配前缀，以该前缀为哈希函数的输入参数，从片外前缀哈希表中查找出其关联的下一跳步信息。片外前缀哈希表是由一组多选择哈希表构成，且每个多选择哈希表存在一组前缀长度相同的前缀规则子集。OET 采用备份方式进行离线更新，确保无中断的 IP 地址查找。

本文采用实际 IP 前缀规则集进行了实验评估。实验结果表明：与已有多叉特里树编码方法相比，OET 存储空间开销显著减少。例如，对于实际 IPv4 和 IPv6 前缀规则集，与树位图特里树相比，OET 在存储空间开销上分别减少了 60%~76%和 55%~63%。因此，OET 是一种存储高效的数据结构，整个 OET 可存储在片上存储器中，从而实现高速 IP 地址查找，满足虚拟路由器和软件路由器的可扩展性要求。

参考文献：

[1] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, “Survey and taxonomy of IP address lookup algorithms,” IEEE Network, vol.15, no.2, pp.8-23, 2001.

[2] M. Caesar, M. Casado, T. Koponen, J. Rexford, and S. Shenker, “Dynamic route computation considered harmful,” ACM SIGCOMM Computer Communication Review, vol.40, no.2, pp.66-71, 2010.

[3] H. Song, F. Hao, M. Kodialam, and T. V. Lakshman, “IPv6 lookups using distributed and load balanced Bloom filters for 100Gbps core router line cards,” in IEEE INFOCOM, 2009, pp.2518-2526.

[4] M. Bando and H. J. Chao. “FlashTrie: hash-based prefrix-compressed trie for IP route lookup beyond 100Gbps,” in IEEE INFOCOM, 2010, pp.1-9.

[5] “BGP table,” http://bgp.potaroo.net.

[6] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford, “In VINI veritas: realistic and controlled network experimentation,” in ACM SIGCOMM, 2006, pp.3-14.

[7] M. B. Anwer and N. Feamster, "Building a fast, virtualized data plane with programmable hardware," ACM SIGCOMM Computer Communications Review, vol.40, no.1, pp.75-82, 2010.

[8] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: exploiting parallelism to scale software routers," in ACM SOSP, 2009, pp.15-28.

[9] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," in ACM SIGCOMM, 2010.

[10] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: power-efficient TCAMs for forwarding engines," in IEEE INFOCOM, 2003, pp.42-52.

[11] K. Zheng, C. Hu, H. Lu, and B. Liu, "A TCAM-based distributed parallel IP lookup scheme and performance analysis," IEEE/ACM Transactions on Networking, vol.14, no.4, pp.863-875, 2006.

[12] W. Lu and S. Sahni, "Low power TCAMs for very large forwarding tables," IEEE Transactions on Networking, vol.18, no.3, pp.948-959, 2010.

[13] A. Border and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in IEEE INFOCOM, 2001, pp.1454-1463.

[14] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, "Longest prefix matching using Bloom filters," in ACM SIGCOMM, 2003, pp.201-212.

[15] J. T. M. Waldvogel, G. Varghese, and B. Plattner, "Scalable high speed IP routing lookups,: in ACM SIGCOMM, 1997, pp.25-36.

[16] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar, "Chisel: a storage-efficient collision-free hash-based network processing architecture," in ISCA, 2006, pp.203-215.

[17] S. Kumar, J. Tuner, P. Crowley, and M. Mitzenmacher, "HEXA: compact data structures for faster packet processing," in IEEE ICNP, 2007, pp.246-255.

[18] H. Yu, R. Mahapatra, and L. Bhuyan, "A hash-based scalable IP lookup using Bloom and fingerprint filters," in IEEE ICNP, 2009, pp.264-273.

[19] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in IEEE INFOCOM, 1998, pp.1240-1247.

[20] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion,: in ACM SIGMETRICS, 1998, pp.1-11.

[21] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in ACM SIGCOMM, 1997, pp.3-14.

[22] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: hardware/software IP lookups with incremental updates," SIGCOMM Computer Communications Review, vol.34, no.2, pp.97-122, 2004.

[23] H. Song, J. Turner, and J. Lockwood, "Shape shift tries for faster IP lookup," in IEEE ICNP, 2005, pp.358-367.

[24] H. Song, M Kodialam, F. Hao, and T. V. Lakshman, "Scalable IP lookups using shape graphs," in IEEE ICNP, 2009, pp.73-82.

[25] J. Hasan and T. N. Vijaykumar, "Dynamic pipelining: making IP lookup truly scalable," in ACM SIGCOMM, 2005, pp.205-216.

[26] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh, "A tree based router search engine architecture with single port memories," in ISCA, 2005, pp.123-133.

[27] W. Jiang and V. K. Prasanna, "Beyond TCAM: an SRAM-based multi-pipeline architecture for terabit IP lookup," in IEEE INFOCOM, 2008, pp.1786-1794.

[28] J. Fu and J. Rexford, “Efficient IP address lookup with a shared forwarding table for multiple virtual routers,” in ACM CoNEXT, 2008.

[29] H. Song, M. Kodialam, F. Hao, and T. V. Lakshman, “Building scalable virtual routers with trie braiding,” in IEEE INFOCOM, 2010, pp.1-9.

[30] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, “Fast hash table lookup using extended Bloom filter: an aid to network processing,” in ACM SIGCOMM, 2005, pp.181-192.

[31] S. Kumar, J. Turner, and P. Crowley, “Peacock hashing: deterministic and updatable hashing for high performance networking,” in IEEE INFOCOM, 2008, pp.101-105.

[32] R. Pagh and F. Rodler, “Cuckoo hashing,” vol.51, no.2, pp.122-144, 2004.

[33] A. Kirsch and M. Mitzenmacher, “The power of one move: hashing schemes for hardware,” in IEEE INFOCOM, 2008. pp. 106-110.

作者简介：
黄　昆：　中国科学院计算技术研究所网络技术研究中心，博士后 huangkun09@ict.ac.cn
谢高岗：　中国科学院计算技术研究所，网络技术研究中心主任，研究员，博士生导师
李彦彪：　湖南大学信息科学与工程学院，硕士生
刘向阳：　密歇根州立大学计算机科学与工程系，助理教授

（ 上接第 69 页 ）

[15] User Mode Linux. UML. http://user-mode-linux.sourceforge.net/

[16] Paul Barham, et al. Xen and the Art of Virtualization. SOSP’03

[17] OpenVZ. http:// wiki.openvz.org/Main_Page

[18] Linux-VServer. http://linux-vserver.org

[19] Linux Container. LXC. http://lxc.sourceforge.net/

[20] Rusty Russell. Virtio: Towards a De-Facto Standard For Virtual I/O Devices. ACM SIGOPS Operating Systems Review. 2008

[21] NetFPGA. http://netfpga.org.

[22] IPerf. http://www.noc.ucf.edu/Tools/Iperf/

作者简介：
贺　鹏：　中国科学院计算技术研究所，网络技术研究中心，博士在读生 hepeng@ict.ac.cn
杨建华：　中科院计算技术研究所，网络技术研究中心，副研究员，
张建华：　中科院计算技术研究所，网络技术研究中心，博士在读生
谢高岗：　中国科学院计算技术研究所，网络技术研究中心主任，研究员，博士生导师